Lists

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - *its position or index*. The first index is zero, the second index is one, and so forth.Like a string, **a list is a sequence of values**. In a string, the values are characters; in a list, they can be ***any type***. The values in **list** are called elements or sometimes ***items.*** There are several ways to create a new list; the simplest is to enclose the elements in square brackets ***([and]):***

List Data type:

A list is a value that contains multiple values in an ordered sequence. The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].

What are the different ways to create a list . Explain.

1. Using pair of square brackets [ ]
2. Using built in function like list(), append() and extend()

1.Creation of List using []

You just need to provide name of the list and initialize it with values. Following is an example of a List in Python :

```
# Creation of List
n = [10, 20, 30, 40]
ls = ['crunchy frog', 'ram bladder', 'lark vomit']
myList = ["The", "earth", "revolves", "around", "sun"]
# Nested List
nl = ['spam', 2.0, 5, [10, 20]]
```

2.Using built in functions

- We can add one item to a list using ***list() , append() method or add several items using extend()*** method.

```
L = list() # empty list
L.append(10)
print(L)
L.extend([2,3,6,7])
print(L)
```

```
[10]
[10, 2, 3, 6, 7]
```

#Example

- cheeses = ['Cheddar', 'Edam', 'Gouda']
- numbers = [17, 123]
- empty = [ ]
- **print(cheeses, numbers, empty)**

Example

- list1 = ['physics', 'chemistry', 1997, 2000]
- list2 = [1, 2, 3, 4, 5]
- list3 = ["a", "b", "c", "d"]
- print(list1)
- print(list2)
- print(list3)

Why Lists are mutable?

Unlike strings, lists are mutable because *you can change the order of items in a list or reassign an item in a list*. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

 # Example for Mutable
numbers = [17, 123]
numbers[1] = 5
print(numbers)
Output -> [17,5]

Examples : Creation of Lists

Example 1: List of numbers

```
1 [1,2,3]
```

```
[1, 2, 3]
```

Example2 : List of Strings

```
1 ['cat','bat','rat','elephant']
```

Example 3: Creation of List of integers

```
1 t = [1,2,3]
```

```
1 t
```

```
[1, 2, 3]
```

Example 4: Creation of List of strings

```
1 t = ['cat','bat','rat','elephant']
```

```
1 t
```

```
['cat', 'bat', 'rat', 'elephant']
```

Creation of Empty list

Example 1:

```
1 []
```

```
[]
```

Example 2:

```
1 t = []
```

```
1 t
```

```
[]
```

Example : Traversing List

```
1 t = ['hello python',23.4567, 22,None,True,False]
2 for i in t:
3     print(i)
```

```
hello python
23.4567
22
None
True
False
```

```
1 "Hello" + t[0]
```

```
'Hellohello python'
```

Concatenation of list elements and string

```
1 "Hello" + t[0]
```

```
'Hellohello python'
```

```
1  spam = ['cat', 'bat', 'rat', 'elephant']
```

```
1 'Hello ' + spam[2]
```

```
'Hello rat'
```

```
1 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
```

```
'The bat ate the cat.'
```

```
1 x = "str1 "  + "str2 " + "str3 " + " str4 "
2 x
```

```
'str1 str2 str3  str4 '
```

```
1 spam = ['cat', 'bat', 'rat', 'elephant']
2 spam[10000]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-33-51564caf19d1> in <module>
      1 spam = ['cat', 'bat', 'rat', 'elephant']
----> 2 spam[10000]

IndexError: list index out of range
```

```
1  spam = ['cat', 'bat', 'rat', 'elephant']
```

```
1 spam[1]
```

```
'bat'
```

```
1 spam[1.0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-a283f69a570a> in <module>
----> 1 spam[1.0]

TypeError: list indices must be integers or slices, not float
```

List Concatenation and Replication

```
1  [1, 2, 3] + ['A', 'B', 'C']
```

```
[1, 2, 3, 'A', 'B', 'C']
```

```
1  ['X', 'Y', 'Z'] * 5
```

```
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
```

```
1  spam = [1,2,3]
```

```
1  spam = spam + ['A','B','C']
2  spam
```

```
['cat', 'aardvark', 'aardvark', 12345, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

```
1  spam = ['cat', 'bat', 'rat', 'elephant']
```

```
1  del spam[2]
```

```
1  spam
```

```
['cat', 'bat', 'elephant']
```

```
1  del spam[2]
```

```
1  spam
```

```
['cat', 'bat']
```

```
1  del spam
```

```
1  spam
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-149-69a5a1000f98> in <module>
----> 1 spam

NameError: name 'spam' is not defined
```

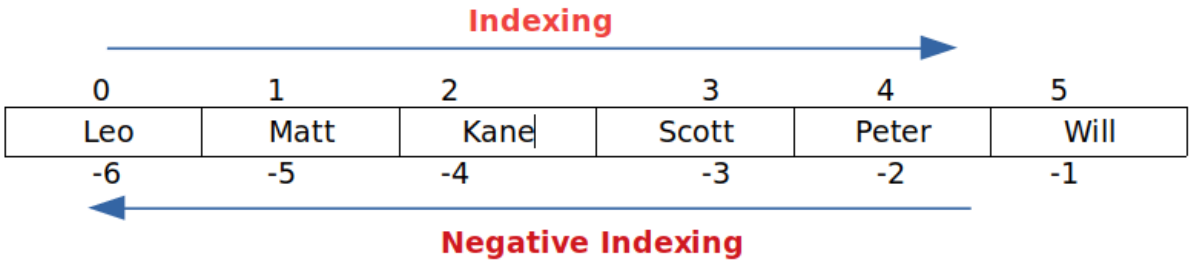
Getting Individual Values in a List with Indexes

Mapping in lists

- You can think of a list as a relationship between *indices and elements*.
- This relationship is called a ***mapping***; each index "***maps to***" one of the elements.

List indices

- List indices work the same way as string indices:
- Any **integer expression** can be used as an index.
- If **you try to read or write an element that does not exist**, you get an Index **Error**.
- If an index has a ***negative value, it counts backward from the end of the list.***



Programming Examples on List Indexing

```
1 t = [ 'hello python',23.4567, 22,None,True,False]
```

```
1 t
```

```
['hello python', 23.4567, 22, None, True, False]
```

```
1 t[0]
```

```
'hello python'
```

```
1 t[1]
```

```
23.4567
```

```
1 t[2]
```

```
22
```

```
1 t[3]
```

```
1 t[4]
```

```
True
```

```
1 t[5]
```

```
False
```

```
1 t[6]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-19-d806090743c9> in <module>
----> 1 t[6]

IndexError: list index out of range
```

```
1 ['cat', 'bat', 'rat', 'elephant'][3]
```

```
'elephant'
```

```
1 ['cat', 'bat', 'rat', 'elephant'][0]
```

```
'cat'
```

```
1 "Hello" + t[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-038643f07208> in <module>
----> 1 "Hello" + t[1]

TypeError: can only concatenate str (not "float") to str
```

in operator in lists

- The **in** operator also works on lists.
- cheeses = ['Cheddar', 'Edam', 'Gouda']
- 'Edam' **in** cheeses True
- 'Brie' **in** cheeses False

```
1 t = ['hello python',23.4567, 22,None,True,False]
2 for i in t:
3     print(i)
```

```
hello python
23.4567
22
None
True
False
```

Negative Indexes

```
1 x = [ 10,20,30,40,50]
```

```
1 x[-1]
```

```
50
```

```
1 x[-1]
```

```
50
```

```
1 x[-2]
```

```
40
```

```
1 x[-3]
```

```
30
```

```
1 x[-4]
```

```
20
```

```
1 x[-5]
```

```
10
```

Explain Traversing a list using a for loop

The most common way to traverse the elements of a list is with a for loop. The syntax and example is as illustrated in the example below :

Example1 :

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
for ch in cheeses:
        print(ch)
```

Example2:

```
numbers = [10,20,30,45,60]
for  i  in range(len(numbers)):
         numbers[i] = numbers[i] * 2
         print(numbers[i])
```

This loop traverses the list and updates each element. ***len*** *returns the number of elements in the list*. **range** returns a list of indices from ***0 to n – 1***, where ***n*** is the length of the list

Note1 : A for loop over an empty list never executes the body:

```
empty = [ ]
for x in empty:
                print('This never happens.')
```

Output :

- **Note2 :** Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:
- **['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]**

With Example Explain the List operations + and *

+ : operator concatenates lists
*****: operator repeats a list a given number of times

+ : operator concatenates lists

Example :

a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
Output :[1, 2, 3, 4, 5, 6]

*: operator repeats a list a given number of times

Example :

- [0] * 4
  Output : [0, 0, 0, 0]

- [1, 2, 3] * 3
- **Output :** [1, 2, 3, 1, 2, 3, 1, 2, 3]

The first example repeats four times. The second example repeats the list three times.

Slicing in List : **Getting Sublists with Slices**

Slice operation is used to slice the given lists into required segments /parts as illustrated below :

t = **['a', 'b', 'c', 'd', 'e', 'f']**
t[1:3] **Output :['b', 'c']**
t[:4] **Output :['a', 'b', 'c', 'd']**
t[3:] **Output :['d', 'e', 'f']**

Note : If you omit the **first** index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

t[:] Output : ['a', 'b', 'c', 'd', 'e', 'f']

- Since lists are mutable, it is often useful to make a copy before performing operations that *fold, spindle, or mutilate lists*. A slice operator on the left side of an assignment can update multiple elements:
  t = ['a', 'b', 'c', 'd', 'e', 'f']
  t[1:3] = ['x', 'y']
  print(t)
  Output : ['a', 'x', 'y', 'd', 'e', 'f']

Programming Examples on List Slicing

```
1  x = [ 10,20,30,40,50]
```

```
1  x[0:2]  # start to end -1  0 - 2-1
```

```
[10, 20]
```

```
1 x[3:5]
```

```
[40, 50]
```

```
1 x[:]
```

```
[10, 20, 30, 40, 50]
```

```
1 x[:-1]
```

```
[10, 20, 30, 40]
```

```
1 x[:-2]
```

```
[10, 20, 30]
```

```
1 x[-5:]
```

```
[10, 20, 30, 40, 50]
```

```
1 x[::-1]
```

```
[50, 40, 30, 20, 10]
```

```
1 x[2::]
```

```
[30, 40, 50]
```

Getting a List's Length with len()

```
1 len(x)
```

```
5
```

Programming Examples on List Slicing

```
1 x[::]
```

```
[10, 20, 30, 40, 50]
```

```
1 x[:]
```

```
[10, 20, 30, 40, 50]
```

```
1 x[::] # start : stop : step
```

```
[10, 20, 30, 40, 50]
```

```
1 x[-1:-6:-1]
```

```
[50, 40, 30, 20, 10]
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like spam = 42. However, you can also use an index of a list to change the value at that index. For example, spam[1] = 'aardvark' means “Assign the value at index 1 in the list spam to the string 'aardvark'.” Enter the following into the interactive shell:

```
1 spam = ['cat', 'bat', 'rat', 'elephant']
```

```
1 spam
```

```
['cat', 'bat', 'rat', 'elephant']
```

```
1 spam[1] ='aardvark'
```

```
1 spam
```

```
['cat', 'aardvark', 'rat', 'elephant']
```

```
1 spam[2] = spam[1]
```

```
1 spam
```

```
['cat', 'aardvark', 'aardvark', 'elephant']
```

```
1 spam[-1] = 12345
```

```
1 spam
```

```
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The * operator can also be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
1  [1, 2, 3] + ['A', 'B', 'C']
```

```
[1, 2, 3, 'A', 'B', 'C']
```

```
1  ['X', 'Y', 'Z'] * 3
```

```
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
```

```
1  spam = [1,2,3]
```

```
1  spam = spam + ['A','B','C']
2  spam
```

```
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del statements

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
1  spam = ['cat', 'bat', 'rat', 'elephant']
```

```
1  del spam[2]
```

```
1  spam
```

```
['cat', 'bat', 'elephant']
```

```
1  del spam[2]
```

```
1  spam
```

```
['cat', 'bat']
```

The del statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If you try to use the variable after deleting it, you will get a NameError error because the variable no longer exists. In practice, you almost never need to delete simple variables. The del statement is mostly used to delete values from lists.

Working With Lists

When you first begin writing programs, it's tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

(I don't actually own this many cats, I swear.) It turns out that this is a bad way to write code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as allMyCats1.py:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' + catName5 + ' ' + catName6)
```

```
Enter the name of cat 1:
abc
Enter the name of cat 2:
def
Enter the name of cat 3:
xyz
Enter the name of cat 4:
ruby
Enter the name of cat 5:
sev
Enter the name of cat 6:
ind
The cat names are:
abc def xyz ruby sev ind
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the allMyCats1.py program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, type the following source code and save it as allMyCats2.py:

```
1  catNames = []
2  while True:
3      print('Enter the name of cat ' + str(len(catNames) + 1) +
4   ' (Or enter nothing to stop.):')
5      name = input()
6      if name == '':
7          break
8      catNames = catNames + [name] # list concatenation
9  print('The cat names are:')
10 for name in catNames:
11     print(' ' + name)
```

```
Enter the name of cat 1 (Or enter nothing to stop.):
zomphy
Enter the name of cat 2 (Or enter nothing to stop.):
ruby
Enter the name of cat 3 (Or enter nothing to stop.):
seembu
Enter the name of cat 4 (Or enter nothing to stop.):
soni
Enter the name of cat 5 (Or enter nothing to stop.):
raavi
Enter the name of cat 6 (Or enter nothing to stop.):
veeru
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
 zomphy
 ruby
 seembu
 soni
 raavi
 veeru
```

Using for Loops with Lists

In Chapter 2, you learned about using for loops to execute a block of code a certain number of times. Technically, a for loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

This is because the return value from range(4) is a list-like value that Python considers similar to [0, 1, 2, 3]. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print(i)

```

```
0
1
2
3
```

What the previous for loop actually does is loop through its clause with the variable i set to a successive value in the [0, 1, 2, 3] list in each iteration. NOTE In this book, I use the term list-like to refer to data types that are technically named sequences. You don't need to know the technical definitions of this term, though. A common Python technique is to use range(len(someList)) with a for loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

```
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using range(len(supplies)) in the previously shown for loop is handy because the code in the loop can access the index (as the variable i) and the value at that index (as supplies[i]). Best of all, range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the in and not in operators. Like other operators, in and not in are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
'howdy' in ['hello', 'hi', 'howdy', 'heyas']
```

```
True
```

```
 spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
'cat' in spam
```

```
False
```

```
 'howdy' not in spam
```

```
False
```

```
'cat' not in spam
```

```
True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as myPets.py:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

```
Enter a pet name:
FootRot
I do not have a pet named FootRot
```

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
name = ['Rohan', 'Sneha', 'Rakshith']
```

```
std1,std2,std3 = name
```

```
print(std1,std2,std3)
```

```
Rohan Sneha Rakshith
```

```
# The number of variables and the length of the list
# must be exactly equal or Python will give you a ValueError
```

```
std1,std2,std3,std4 = name
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-e9d018e2cf0e> in <module>
----> 1 std1,std2,std3,std4 = name

ValueError: not enough values to unpack (expected 4, got 3)
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

```
1  x = 72
2  x = x+1
3  x
```

```
73
```

As a shortcut, you can use the augmented assignment operator += to do the same thing:

```
1  x = 72
2  x+=1
3  x
```

```
73
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table 4-1

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

The += operator can also do string and list concatenation, and the *= operator can do string and list replication. Enter the following into the interactive shell:

```
1 spam = 'Hello'
2 spam+='world'
3 spam
```

```
'Helloworld'
```

```
1 bacon =['Zophie']
2 bacon*=3
3 bacon
```

```
['Zophie', 'Zophie', 'Zophie']
```

Methods

A method is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in spam, you would call the index() list method (which I’ll explain next) on that list like so: spam.index('hello'). The method part comes after the value, separated by a period. Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a ValueError error. Enter the following into the interactive shell:

```
1 spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
1 spam.index('hello')
```

```
0
```

```
1 spam.index('heyas')
```

```
3
```

```
1 spam.index('howdy howdy howdy')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-61-aa717b951a7e> in <module>
----> 1 spam.index('howdy howdy howdy')

ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that index() returns 1, not 3:

```
spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
```

```
spam.index('Pooka')
```

```
1
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the append() and insert() methods. Enter the following into the interactive shell to call the append() method on a list value stored in the variable spam:

```
num = ["one","two","three"]
```

```
num.append("four")
```

```
num
```

```
['one', 'two', 'three', 'four']
```

The previous append() method call adds the argument to the end of the list. The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
num = ["one","two","three"]
```

```
num.insert(1,'chicken')
```

```
num
```

```
['one', 'chicken', 'two', 'three']
```

Notice that the code is spam.append('moose') and spam.insert(1, 'chicken'), not spam = spam.append('moose') and spam = spam.insert(1, 'chicken'). Neither append() nor insert() gives the new value of spam as its return value. (In fact, the

return value of append() and insert() is None, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified in place. Modifying a list in place is covered in more detail later in "Mutable and Immutable Data Types" on page 94. Methods belong to a single data type. The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the AttributeError error messages that show up:

```
1 name = "hello"
```

```
1 name.append("python")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-71-841e03043bd7> in <module>
----> 1 name.append("python")

AttributeError: 'str' object has no attribute 'append'
```

```
1 no = 42
```

```
1 no.insert(1,'world')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-72-e65efc34d13b> in <module>
----> 1 no.insert(1,'world')

NameError: name 'no' is not defined
```

Removing Values from Lists with remove()

The remove() method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
1 animals = ['cat', 'bat', 'rat', 'elephant']
```

```
1 animals.remove('bat')
```

```
1 animals
```

```
['cat', 'rat', 'elephant']
```

```
1 animals.remove('chicken')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-83-f2fce26c25e9> in <module>
----> 1 animals.remove('chicken')

ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
1 name = ['raj','ravi','seenu','raj','rakshit','raj']
```

```
1 name.remove('raj')
```

```
1 name
```

```
['ravi', 'seenu', 'raj', 'rakshit', 'raj']
```

The del statement is good to use when you know the index of the value you want to remove from the list. The remove() method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the sort() method. For example, enter the following into the interactive shell:

```
1 x = [2, 5, 3.14, 1, -7]
```

```
1 x.sort()
```

```
1 x
```

```
[-7, 1, 2, 3.14, 5]
```

```
1 x = ['apple','banana','watermelon','grapes']
2 x.sort()
3 x
```

```
['apple', 'banana', 'grapes', 'watermelon']
```

You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order. Enter the following into the interactive shell:

```
1 x.sort(reverse = True)
```

```
1 x
```

```
['watermelon', 'grapes', 'banana', 'apple']
```

There are three things you should note about the sort() method. First, the sort() method sorts the list in place; don't try to capture the return value by writing code like spam = spam.sort().Second, you cannot sort lists that have both number values and string values in them, since Python doesn't know how to compare these values. Type the following into the interactive shell and notice the TypeError error:

```
1 y = [1, 3, 2, 4, 'Anaha', 'Boby']
2 y.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-93-30b50c90903f> in <module>
      1 y = [1, 3, 2, 4, 'Anaha', 'Boby']
----> 2 y.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

Third, sort() uses "ASCIIbetical order" rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase a is sorted so that it comes after the uppercase Z. For an example, enter the following into the interactive shell

```
1 name = ['Alice','ants',"Boby","badgergaurd",'Carol','Cats']
2 name.sort()
3 name
```

```
['Alice', 'Boby', 'Carol', 'Cats', 'ants', 'badgergaurd']
```

If you need to sort the values in regular alphabetical order, pass str. lower for the key keyword argument in the sort() method call.

```
1 sym = [ 'a','z','A','Z']
2 sym.sort(key = str.lower)
3 sym
```

```
['a', 'A', 'z', 'Z']
```

This causes the sort() function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical elif statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as magic8Ball2.py.

```
import random
messages = ['It is certain',
 'It is decidedly so',
 'Yes definitely',
 'Reply hazy try again',
 'Ask again later',
 'Concentrate and ask again',
 'My reply is no',
 'Outlook not so good',
 'Very doubtful']
print(messages[random.randint(0, len(messages) - 1)])
```

```
It is certain
```

Additional Notes on lists:

Explain the following List methods with examples :

a. **Append**
b. **Extend**
c. **Sort**
d. **Delete**

a. Append

- Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

```
1 t = ['a', 'b', 'c']
2 t.append('d')
3 print(t)
```

```
['a', 'b', 'c', 'd']
```

b.Extend

extend takes a list as an argument and appends all of the elements:

```
1 t1 = ['a', 'b', 'c']
2 t2 = ['d', 'e']
3 t1.extend(t2)
4 print(t1)
```

```
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

c.Sort

- **sort** arranges the elements of the list from low to high:

```
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

Note : Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

e. Deleting elements

There are several ways to delete elements from a list.

If you know the ***index of the element you want***, you can **use pop**:

```
t = ['a', 'b', 'c']
x = t.pop(1)
print(t)
print(x)

```

```
['a', 'c']
b
```

Note : pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

Deleting using del

- If you **don't need the removed value**, you can use the del operator:

```
t = ['a', 'b', 'c']
del  t[1]
print(t)

```

```
['a', 'c']
```

Deleting using remove

If you know the element you want to remove (**but not the index**), you can use remove:

```
t = ['a', 'b', 'c']
t.remove('b')
print(t)
```

Output :['a', 'c']

The return value from remove is None.

Deleting and Slicing

To remove more than one element, you can use ***del with a slice index***:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
del t[1:5]
print(t)
```

Output :['a', 'f']

As usual, the ***slice*** selects all the elements up to, but not including, the second index.

Lists and functions

There are a number of**built-in functions that can be used on lists** that allow you to quickly look through a list without writing your own loops:

```
>>>nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
```

```
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

Note

- The **sum()** function only works when the list elements are numbers.
- The other functions (**max(), len(), etc**.) work with lists of strings and other types that can be comparable.

Programming Examples : To find the average of a given number with list and without list

First, the program to compute an average without a list

```
# Program to compute without a list
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done':break
    value = float(inp)
    total = total + value
    count = count + 1
average = total / count
print('Average:', average)
```

Program using list functions

```
numlist = list()

while (True):
    inp = input('Enter a number: ')
    if inp == 'done':break
    value = float(inp)
    numlist.append(value)
average = sum(numlist) / len(numlist)
print('Average:', average)
```

```
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: done
Average: 3.0
```